SOLVING VERY LARGE, SPARSE LINEAR SYSTEMS ON MESH-CONNECTED PARALLEL COMPUTERS

Torstein Opsahl*

MRJ, Inc.
Perkin-Elmer Advanced Development Center
Oakton, Virginia 22124

John Reif**

Duke University
Durham, North Carolina 27706

ABSTRACT

In this paper, the implementation of Pan and Reif's Parallel Nested Dissection algorithm on mesh-connected parallel computers is described. This is the first known algorithm that allows very large, sparse linear systems of equations to be solved efficiently in polylog time using a small number of processors. We describe how the processor bound of PND can be matched to the number of processors available on a given parallel computer by slowing down the algorithm by constant factors. Also, for the important class of problems where G(A) is a grid graph, we detail a unique memory mapping that reduces the inter-processor communication requirements of PND to those that can be executed on mesh-connected parallel machines. The paper concludes with a description of an implementation on the Goodyear Aerospace Massively Parallel Processor (MPP), located at NASA Goddard Space Flight Center, for which we give a detailed discussion of data mappings and performance issues.

Keywords: Parallel Computation, Parallel Nested Dissection, Recursive Factorization, Mesh-connected Network, Separator, Grid Graph.

* Previous association, Science Applications Research, Lanham, Maryland.

** This work was supported by Office of Naval Research Contract N00014-80-C-0647 and NSF grant DCR-850351

INTRODUCTION

The solution of large, sparse linear systems, Ax=b, pervades many areas of physics and engineering. Although parallel algorithms for tackling these problems have existed for a number of years, they have usually been impractical to implement because of unrealistically high time bounds or processor bounds, or both. In other cases, numerical stability has been a problem; that is, unless the calculations were performed in infinite precision they would yield no solution at all. However, none of the above problems apply to Pan and Reif's (Ref. 8) parallel nested dissection (PND) algorithm. This algorithm is based on computing a special recursive factorization of A, thus reducing the problem of inverting a large sparse matrix to that of inverting a number of much smaller dense matrices. PND has a considerably smaller processor bound than other polylog parallel methods for solving sparse linear systems. Furthermore, for any given parallel computer with p processors, the algorithm can be slowed down to give the best known time bound for this processor bound p.

If A is an nxn matrix, we define a graph G(A)=(V,E) to be the undirected graph with vertex set V={1,...,n} and edge set E={{i,j}|A_{ij} not equal to 0}. Here we shall confine our attention to the application of PND to sparse, linear systems where G(A) is a two-dimensional (2-D) grid graph. Such systems occur very extensively in the solution of partial differential equations. However, it should be emphasised that the algorithm is not restricted to this class of problems.

The Massively Parallel Processor (MPP) is an SIMD controlled, fine-grained, 2-D mesh-connected parallel computer with 16,384 Processing Elements (PEs) built by Goodyear Aerospace for NASA. This paper will describe how PND can be implemented on this computer and other parallel machines with a similar architecture. The implementation is particularly challenging because of the SIMD control structure of these computers. Central to the implementation is a unique memory mapping that reduces the communication requirements of PND to those that can be executed using only mesh connections. Also, the algorithm has been implemented on the Connection Machine, built by Thinking Machines Corporation, whose processors are connected in a hypercube. This implementation is described in Ref. 5.

In the next section, we give a more detailed description of PND, while the final section contains the implementation method for mesh-connected parallel computers together with some performance estimates.

DESCRIPTION OF PND FOR GRID GRAPHS

The method of parallel nested dissection (PND) and its proof has been described in detail elsewhere (Ref. 8). Here, we will give a relatively self-contained overview of the method. Since it is far from obvious how PND can be used on the MPP, the description concentrates on those aspects of the method that are critical to understanding our implementation for mesh-connected parallel computers.

Fundamental to the nested dissection of undirected graphs is the idea of separators. A separator is a set of vertices that partitions a graph into two sub-graphs that are connected only through the separator. Each of the sub-graphs must contain no more than 2/3 and no less than 1/3 of the nodes of G(A). For a graph of n vertices, the size of any separator is bound by a function s(n). In the case of planar graphs s(n) is $O(n^{1/2})$. An undirected graph is said to have an s(n)-separator family if the class of all its sub-graphs has an s(n)-separator family. Binary trees, for instance, have a 1-separator family while a d-dimensional grid of uniform size in each dimension has $n^{1-(1/d)}$ -separators.

The removal of separators from a graph, G(A), and its resultant sub-graphs, is in fact analogous to eliminating the unknowns of the associated linear system, in the order given by the numbering of the vertices in the separators. Sequential methods for solving sparse, linear systems have long made use of nested dissection (Ref. 3) to reduce the problem to that of inverting dense matrices of size at most s(n)xs(n). PND is the first algorithm for efficiently solving sparse, positive definite linear systems, Ax=b, on parallel computers that only requires polylog time and $s(n)^3$ processors. Moreover, the algorithm is numerically stable. PND constructs in 0(logn) stages a special recursive factorization of A. This factorization is distinct from the LDL^T-factorization used in sequential dissection. Another important difference between sequential dissection and PND is that the latter requires the elimination of many, rather than one, separators at each step of the algorithm.

Suppose G(A) has a separator that decomposes the graph into two sub-graphs. Furthermore, suppose that these sub-graphs can be partitioned recursively through the use of separators. In this way we can construct a separator tree (see figure 1) whose root is the separator of G(A). This root has two children that are the separators of the two sub-graphs. Each of these children is in turn the parent of two new children and so on. The leaves of the separator tree are singleton node sets of remaining sub-graphs.

One of the general difficulties of PND is to compute efficiently in parallel s(n)-separators. Fortunately, for the practically important case of grid graphs, which we are concerned with here, there is a simple way to do this in O(logn) steps. At the completion of these steps the vertices of the graph have been renumbered from 1 to n using the separator tree as a guide. PND requires that A is permuted to reflect this renumbering. Assuming this has been done, Pan and Reif (Ref. 8) define a recursive s(n)-factorization by a sequence of matrices A_0, A_1,..., A_d where,

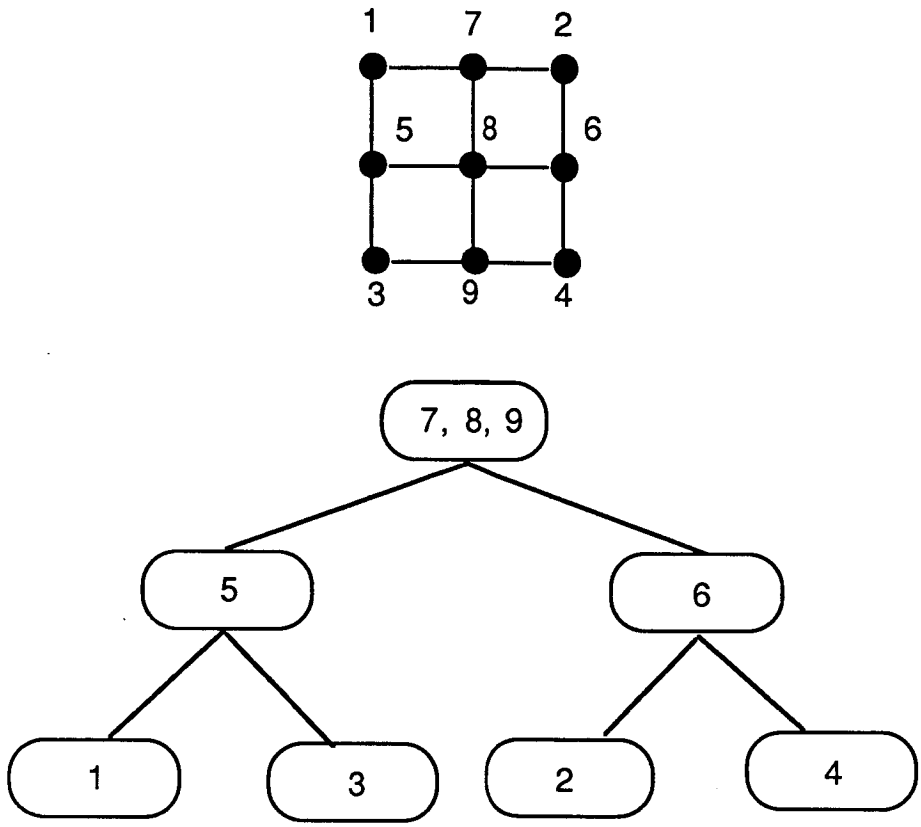

Figure 1 - Separator tree for a 3x3 grid graph

$A_0 = PAP^T$; P is a permutation matrix

$$A_h = \begin{bmatrix} X_h & Y_h^T \\ Y_h & Z_h \end{bmatrix} \qquad (1)$$

$$Z_h = A_{h+1} + Y_h X_h^{-1} Y_h^T;\ h=0,1,...,d-1$$

There are several important points to note about equation (1). Let d be O(logn). Also, let N_h be the number of separators at level h and $n_{h,k}$ ($k= 1...N_h$, $n_{h,k} <= (2/3)^h$) the maximum size of any separator at the h^{th} level. It can then be seen that X_h is a block-diagonal matrix consisting of N_h square blocks of sizes at most $n_{h,k} x n_{h,k}$. Thus, X_h^{-1} can be computed by inverting these small diagonal blocks separately. It also follows that the matrix product $Y_h X_h^{-1} Y_h^T$ can be decomposed into $O(N_h)$ triplets of sizes $O(n_{h,k} x n_{h,k})$. Pan and Reif (Ref. 8) prove that both of the above operations can be performed in $O(\log^2 s(n))$ time using at most $s(n)^3$ processors. In fact, the main advantage of the recursive s(n)-factorization is that it only requires O(logn) stages. This implies that the entire s(n)-factorization of A can be computed in $O(\log n(\log^2 s(n))$ time using $s(n)^3$ processors. However, on even the most parallel of currently available computers such as the MPP and the Connection Machine, this bound will severely restrict the size of problems that can be solved. Fortunately, the PND algorithm can easily be slowed

down so that the number of processors required matches that of a specific parallel machine. In particular, if $s(n)=n^{1/2}$ then n processors is sufficient provided the algorithm's speed is reduced by an $n^{1/2}$ factor. As a matter of fact, a careful time analysis shows that the total time to complete the s(n)-recursive factorization of A in this case is $O(n^{1/2})$. As we shall see in the next section this allows very large sparse matrices to be inverted on the MPP.

Finally, the recursive s(n)-factorization allows us to write

$$A_h = \begin{bmatrix} I & 0 \\ Y_h X_h^{-1} & I \end{bmatrix} \begin{bmatrix} X_h & 0 \\ 0 & A_{h+1} \end{bmatrix} \begin{bmatrix} I & X_h^{-1} Y_h^T \\ 0 & I \end{bmatrix}$$

--- (2)

and hence

$$A_h^{-1} = \begin{bmatrix} I & -X_h Y_h^T \\ 0 & I \end{bmatrix} \begin{bmatrix} X_h^{-1} & 0 \\ 0 & A_{h+1}^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ Y_h X_h^{-1} & I \end{bmatrix}$$

---- (3)

Thus, given an s(n)-factorization of A, it is easy to recursively compute $A^{-1}b$ for any column vector b of length n. This "backsolving" computation can be performed in time O(lognlogs(n)) using $s(n)^2$processors. It should be noted that here we do not have to slow down the parallel computation when $s(n)=n^{1/2}$ and the number of available processors is n.

In summary, a step in the recursive s(n)-factorization of A is accomplished by moving one level up in the separator tree removing the level h separators from the sub-graphs by eliminating the corresponding unknowns of A_{h+1}. This involves $O(N_h)$ matrix operations on matrices of size $O(n_{h,k}^2)$ (see above). If these operations are performed by systolic algorithms, they require a maximum of $s(n)^2$ processors and take O(s(n)) time. Once the root of the tree is reached ($N_h=1$, $h=d-1$), the s(n)-factorization is complete. Two traversals of the separator tree are then necessary to perform the "backsolving" because of the recursive nature of equation (3) and the total time for these computations is O(lognlogs(n)).

IMPLEMENTATION OF PND ON THE MPP

In this section we will discuss an implementation of PND, for the case where G(A) is a $n^{1/2}xn^{1/2}$ grid graph, for mesh-connected parallel computers, in general, and for the MPP in particular.

Representing a Grid Graph as a Set of Boxes

To be able to fully explain our implementation of PND on the MPP, we first need to expand on some of the ideas of the previous section. Consider a phase h of the recursive factorization where X_h is a block-diagonal matrix consisting of a total of N_h blocks $x_{h,k}$ ($k=1..N_h$) of size at most $n_{h,k}xn_{h,k}$. Each of these diagonal blocks can be associated with a given separator. The same applies to the triplets $y_{h,k}(x_{h,k})^{-1}(y_{h,k})^T$ (size $O(n_{h,k}^2)$ of $Y_h X_h^{-1} Y_h^T$. Furthermore, their existence implies that Z_h can be broken up into N_h blocks $z_{h,k}$ also of size $O(n_{h,k}^2)$. This in turn means that A_h can be considered as being made up of N_h smaller matrices $a_{h,k}$. Each of these will consist of a $x_{h,k}$, $y_{h,k}$, $(y_{h,k})^T$ and a $z_{h,k}$. The crucial observation, however, is that the $m_{h,k}$'s (m= a, x, y, z) are much smaller and denser than the big matrices to which they belong. Thus, a key element of the MPP implementation of PND is to compute A_{h+1} as N_{h+1} a_{h+1}'s. We will now describe in detail our implementation of the h^{th} stage of PND where A_{h+1} must be computed from A_h.

In figure 2, the grid graph of figure 1 has been extended so that each vertex is the junction of four edges, each of unit length. The resultant graph may be represented as a tiling of 2-D boxes with the

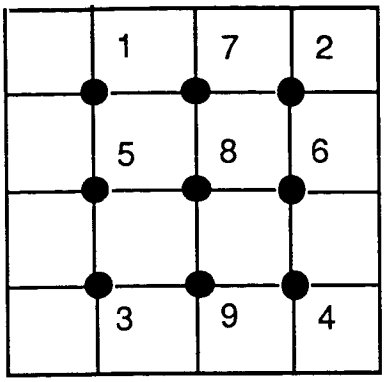

Figure 2 - 3x3 grid graph

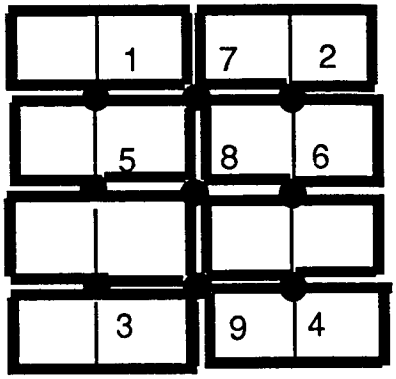

Figure 3 - Box representation of grid graph

vertices lying along their perimeters (see figure 3). Each box is a $2^h x 2^{h+1}$ rectangle or a $2^h x 2^h$ square depending on whether h is odd or even, respectively. Now, in computing the $a_{1,k}$'s (h= 0), the level zero separators are removed (vertices 1, 2, 3 and 4 for the grid graph of figure 3). This may be viewed as merging pairs of boxes into single boxes. All vertices strictly internal to a box will be eliminated by the s(n)-factorization of A. The box representation of a unit edge length grid graph can be used at all levels in the separator tree and its traversal therefore translates into repeated box-compositions. A box $b_{h,k}$ has $2(n_{h,k} + n_{h-1,k} + 2)$ vertices on its perimeter. A matched pair of level h boxes is defined as two boxes that have a common separator $s_{h,k}$. The first vertex on either side of a separator will be referred to as contact points denoted by c_0 and c_1, respectively.

The arithmetic associated with the box-composition breaks down as follows: To compute the $a_{h+1,k}$'s will require N_h dense matrix inversions (or LDL^T decompositions), $2N_h$ matrix multiplications and N_h matrix subtractions. These operations will be performed by systolic algorithms using sub-arrays of $O((n_{h,k})^2)$ processors (see below for alternative non-systolic algorithms for the MPP). Because the two $a_{h,k}$'s of a pair of boxes are not completely disjoint due to the boxes' common edges it will also be necessary to merge pairs of $a_{h,k}$'s by adding together the common edge coefficients in order to compute the $z_{h,k}$, $y_{h,k}$, $x_{h,k}$ and $y_{h,k}^T$ needed in the calculation of $a_{h+1,k}$. This requires another N_h matrix additions. It should be noted that the final stage (h=d-1) of the s(n)-factorization of A involves the inversion of a single matrix of size s(n)xs(n) and that the time required for this is O(s(n)). Hence, the cost of the final stage is the dominant element in the total time for the factorization.

The arithmetic described above will require inter-processor communications. The patterns of these, as well as their cost, depend on how A and its factors are mapped onto the local memories of the PEs. In the following a unique memory mapping scheme is described that creates local communication patterns that are supported by mesh-connected machines such as the MPP. This scheme also ensures that data can be sent to the correct processors using SIMD control so that no explicit address calculations are required.

Mapping a Box Representation of a Grid Graph onto the MPP.

Let each box of a box representation of a grid graph correspond to a $(n_{h,k}+ n_{h-1,k}+ 2)^2$ neighborhood of processors such that the coefficients of adjacent boxes in the graph will be stored in adjacent neighborhoods of PEs. Within each of these neighborhoods, data is laid out using the following rule: The ordering of the coefficients should be that given by taking the vertices of the perimeter of the box in a clock-wise direction, starting in the lower left hand corner, just above the corner vertex. Figure 4 shows the result of using our memory mapping scheme for the pair of boxes of figure 3 with

PE : i ->

j ↓

North Box

	(-)	(-)	(-)	(7)	(1)	(-)
(-)	0	0	0	0	X	0
(-)	0	0	0	0	X	0
(-)	0	0	0	0	0	0
(7)	0	0	0	X	X	0
(1)	0	0	0	0	0	0
(-)	0	0	0	0	0	0

South Box

	(-)	(1)	(7)	(8)	(5)	(-)
(-)	0	0	X	0	X	0
(1)	0	X	X	0	X	0
(7)	0	X	X	X	0	0
(8)	0	0	X	X	X	0
(5)	0	X	0	X	X	0
(-)	0	0	0	0	0	0

X - non-zero coefficient

(m) - vertex number

(-) - missing vertex

Figure 4 - Example of memory mapping for a pair of boxes with common edge (1,7) of 3x3 grid graph of Figure 1.

common edge (1, 7). On a mesh-connected computer with p processors, $(p^{1/2}/(n_{h,k}+n_{h+1,k}+2))^2$ boxes can be operated on in parallel. The actual location of each box within the processor array must be precomputed for all levels of the separator tree before the recursive decomposition of A can start. Our box-compositon strategy greatly simplifies this problem since it allows the processor in which each coefficient is to be stored to be determined based simply on the connectivity of the graph.

Two additional ideas are essential to the successful use of the memory mapping scheme described above. First, it is helpful to think of the boxes as having an orientation: For even h, the separators are horizontal and the boxes lie along a north-south axis while for odd h the separators are vertical so that the boxes have an east-west orientation. Second, the merging of a pair of $a_{h,k}$'s (see above) is facilitated by dividing the vertices on the perimeters of the boxes into sequences that will contain different vertices depending on the position of a box in a pair.

Let $s_{h,k}$ be a separator at the h^{th} level in the separator tree. Then, for a North (East) box let sequence a (sqa) be the $(2n_{h-1,k}+n_{h,k}+1)$ first vertices and sequence b (sqb) the c_1, $s_{h,k}$(in reversed order) and c_0 vertices. Also, for a South (west) box let sequence a (sqa) be the $n_{h-1,k}$ first vertices, sequence b (sqb) the c_0, $s_{h,k}$ and c_1 vertices and sequence c (sqc) the remaining vertices. The steps involved in merging pairs of $a_{h,k}$'s can now be described as:

(i) For each north (east) box reverse the coefficients associated with the contact points about the data for the separators and move c_0 data to the processors containing this data for the south (west) box

(ii) For each north (east) box, reverse the order of the separator coefficients

(iii) Insert the data for sqa and sqb (minus the c_0 coefficients) for the north (east) box after the data for the sqa of the south (west) box: this automatically aligns the c_1 and $s_{h,k}$ coefficients for the two boxes

(iv) do matrix addition

The data movements of (i), (ii) and (iii) translate into shifts through the mesh-connected network. The important points are that all shift distances are given by the dimensions of the boxes and that the data for common edges automatically line up. Having merged the pairs of a_h's, the computation of $a_{h+1,k}$ becomes straightforward since all the required data is now stored in a local neighborhood of processors.

"Backsolving" and Performance Issues

PND reduces the problem of inverting (LDL^T decomposition) a large sparse matrix, A, to that of inverting a number of much smaller dense matrices by computing a recursive s(n)-factorization of A. Now, a choice has to be made as to which algorithms should be used for matrix inversion(or LDL^T decomposition) and multiplication on the MPP. We have implemented both systolic and non-systolic algorithms for these operations. In the case of inversion, a comparison was made between a non-systolic Gauss-Jordan and a systolic Givens rotation method (Ref. 1) This comparison showed that although the latter required much less inter-processor communications than the former, this was off-set by the MPP's broadcasting facility and the smaller number of arithmetic operations of Gauss-Jordan. A similar result was obtained for the matrix multiplication algorithms.

Finally, let us consider the "backsolving" part of PND. An inspection of equation (3) shows that in order to minimize the total amount of arithmetic required to solve a sparse, linear system using PND, some of the quantities computed during the factorization of A should be saved for the "backsolving" calculation. These include $x_{h,k}^{-1}y_{h,k}^T$ and $x_{h,k}^{-1}$. Since each processor of the MPP has only 1k-bits of local memory, it would be impossible to store quantities where they are computed. However, the processors also have access to 16k-bits of staging memory each. This gives the MPP an impressive total amount of storage and makes possible the saving of quantities described above. The overhead of moving data between local processor memories and the staging memory would be less than that incurred if quantities had to be recomputed during the "backsolving" stage.

The remaining issue is now how to perform the sparse multiplication $A^{-1}b$ given our memory

mapping. There are two possible approaches - either to divide b into segments based on the vertices of the perimeters of a box or to store b as a single global data structure. We have chosen to implement the former strategy. Note that in principle, the sum-or tree of the MPP can be used to pipeline the solutions to many sparse linear systems, Ax=b, assuming A is the same for all of them. In particular, we can solve O(s(n)) systems with distinct vectors b in total time O(s(n)) using s(n)2 processors.

With the implementation of PND outlined in this section, and given the storage constraints of the MPP, estimates show that linear systems with up to 16k unknowns can be recursively factorized in about 40 seconds. The "backsolving" will take about 1 second. It should be noted that the size of the problems that can be tackled by our implementation of PND for mesh-connected parallel computers is only limited by available machine memory.

REFERENCES

1. Bojanczyk, A., R. P. Brent and H. T. Kung, "Numerically stable solution of dense systems of linear equations using mesh-connected processors," Technical Report, CMU-CS-81-119, Computer Sci. Dept., Carnegie-Mellon University, Pittsburgh, PA (1981).

2. Csanky, L., "Fast Parallel Matrix Inversion Algorithms," SIAM J. on Computing 5(4), 618-623 (1976).

3. George, J. A., "Nested Dissection of a Regular Finite Element Mesh," SIAM J. on Numerical Analysis 10(2), 345-367 (1973).

4. Hageman, L. and D. Young, "Applied Iterative Methods," Academic Press, New York (1981).

5. Leiserson, C. E., Jill P. Mesirov, Lena Nekludova, S. Omohundro and J. Reif, "Solving Sparse Systems of Linear Equations on the Connection Machine," Annual SIAM Conference, Boston, MA (1986).

6.Lipton, R., D. Rose and R. E. Tarjan, "Generalized Nested Dissection", SIAM J. on Appl. Math. 36, 177-189 (1979).

7. Pan, V., "Fast and Efficient Parallel Algorithms for the Exact Inversion of Integer Matrices," Technical Report, TR 85-2, Computer Sci. Dept., SUNYA, Albany, N. Y. (1985).

8. Pan, V. and J. Reif, "Efficient Parallel Solution of Linear Systems," Technical Report, TR-02-85, Center for Research in Computer Technology, Aiken Computation Laboratory, Harvard University, Cambridge, MA (1985).

9. Pan, V. and J. Reif, "Fast and Efficient Algorithms for Linear Programming and for the Linear Least Squares Problem," Technical Report TR-11-85, Center for Research in Computer Technology, Aiken Computation Laboratory, Harvard University, Cambridge, MA(1985).

10. Pan, V. and J. Reif, "Extension of the Parallel Nested Dissection Algorithm to Path Algebra Problems," Technical Report 85-9, Computer Sci. Dept., SUNYA, Albany, N. Y. (1985).